PHASE1-1.2 PART 1: AWS Cost Collector - Code Generation

OptiInfra Development Series
Phase: Cost Agent (Week 2-3)
Component: AWS Cost Metrics Collection
Estimated Time: 25 minutes setup + 20 minutes validation
Dependencies: P-01 (Bootstrap), 1.1 (Cost Agent Skeleton), 0.10 (Shared Utilities)

📋 Overview

This prompt creates a complete AWS cost collection system that integrates with AWS Cost Explorer API to gather EC2, RDS, Lambda, and other service costs. It identifies optimization opportunities and stores metrics in ClickHouse for analysis.

🎯 Objectives

By the end of this prompt, you will have:

1. ✅ AWS Cost Explorer integration with boto3
2. ✅ EC2, RDS, Lambda, S3, and EBS cost collectors
3. ✅ Automated cost analysis and anomaly detection
4. ✅ Idle resource identification
5. ✅ Cost metrics stored in ClickHouse
6. ✅ Comprehensive unit and integration tests
7. ✅ Prometheus metrics for AWS cost collection

🔧 Prerequisites

Before starting, ensure:

- ☐ PHASE1-1.1 completed - Cost Agent Skeleton running
- ☐ AWS account with Cost Explorer API enabled
- ☐ AWS credentials configured (access key + secret key)
- ☐ boto3 library installed
- ☐ ClickHouse running with cost metrics tables
- ☐ At least 2 weeks of AWS cost data available

📝 Detailed Windsurf Prompt

Create a complete AWS cost collection system for OptiInfra Cost Agent.

CONTEXT:
- Multi-agent LLM infrastructure optimization platform
- Cost Agent already has skeleton (FastAPI, LangGraph, metrics)
- Need to collect real AWS cost data via Cost Explorer API
- Analyze EC2, RDS, Lambda, S3, EBS costs
- Identify idle resources and optimization opportunities
- Store metrics in ClickHouse for time-series analysis
- Project root: ~/optiinfra/services/cost-agent

REQUIREMENTS:

1. AWS COST COLLECTOR BASE (src/collectors/aws/)
   Create base AWS collector class:
   - __init__.py - Module initialization
   - base.py - Base AWS collector with common functionality
     * AWS session management with boto3
     * Credential handling (env vars, IAM roles)
     * Region selection and multi-region support
     * Retry logic with exponential backoff
     * Error handling and logging
     * Connection pooling
     * Rate limit handling (Cost Explorer: 400 req/hour)

   Base class should include:
   - get_session() - Returns configured boto3 session
   - get_client(service_name) - Returns boto3 client with retry config
   - paginate_results() - Helper for paginated API calls
   - handle_throttling() - Automatic retry with backoff
   - log_api_call() - Track API usage for rate limiting

2. COST EXPLORER CLIENT (src/collectors/aws/cost_explorer.py)
   Create Cost Explorer wrapper:
   - CostExplorerClient class
   - get_cost_and_usage() - Main cost data retrieval
     * Date range: configurable (default: last 30 days)
     * Granularity: DAILY, MONTHLY
     * Metrics: UnblendedCost, AmortizedCost
     * Group by: SERVICE, USAGE_TYPE, INSTANCE_TYPE
   - get_cost_forecast() - 30-day cost prediction

- get_savings_plans_utilization() - SP usage metrics
- get_reservation_utilization() - RI usage metrics
- get_right_sizing_recommendations() - AWS native recommendations
- Data transformation to OptiInfra format

Response format:
```
{
  "time_period": {"start": "2025-10-01", "end": "2025-10-31"},
  "total_cost": 120000.50,
  "by_service": {
    "AmazonEC2": 85000.00,
    "AmazonRDS": 20000.00,
    "AWSLambda": 8000.50
  },
  "by_region": {...},
  "by_tag": {...}
}
```

3. EC2 COST COLLECTOR (src/collectors/aws/ec2.py)
Create EC2-specific collector:
- EC2CostCollector class
- collect_instance_costs() - Per-instance cost breakdown
- identify_idle_instances() - CPU <5%, Network <1MB/day
- identify_underutilized_instances() - CPU <20% for 14+ days
- get_spot_opportunities() - Instances eligible for spot
- get_reserved_instance_recommendations() - RI purchase suggestions
- analyze_instance_types() - Usage patterns by instance type
- get_ebs_costs() - EBS volume costs (attached + unattached)
- identify_unattached_ebs() - Volumes not attached to instances

Use CloudWatch for utilization metrics:
- CPU utilization (14-day average)
- Network in/out (14-day average)
- Disk read/write ops

Output format:
```
{
  "instance_id": "i-1234567890abcdef0",
  "instance_type": "m5.2xlarge",
  "region": "us-east-1",
  "monthly_cost": 250.00,
```

```
  "utilization": {
    "cpu_avg": 15.5,
    "network_mb_day": 500.0
  },
  "optimization": {
    "is_idle": false,
    "is_underutilized": true,
    "spot_eligible": true,
    "rightsizing_recommendation": "m5.xlarge",
    "estimated_savings": 125.00
  }
}
```

4. RDS COST COLLECTOR (src/collectors/aws/rds.py)

Create RDS-specific collector:
- RDSCostCollector class
- collect_rds_costs() - Per-instance costs
- identify_idle_databases() - Connection count = 0
- analyze_storage_costs() - Storage, IOPS, backup costs
- get_reserved_instance_recommendations() - RDS RIs
- identify_multi_az_opportunities() - Convert to single-AZ
- analyze_snapshot_costs() - Old snapshot cleanup

CloudWatch metrics:
- DatabaseConnections (14-day average)
- CPUUtilization
- FreeableMemory
- ReadIOPS / WriteIOPS

5. LAMBDA COST COLLECTOR (src/collectors/aws/lambda_costs.py)

Create Lambda-specific collector:
- LambdaCostCollector class
- collect_lambda_costs() - Per-function costs
- analyze_invocations() - Invocation count and duration
- identify_over_provisioned() - Memory > needed
- calculate_optimal_memory() - Cost-performance optimization
- identify_cold_starts() - Functions with high cold start %

CloudWatch metrics:
- Invocations
- Duration

- Errors
- Throttles

6. S3 COST COLLECTOR (src/collectors/aws/s3.py)
Create S3-specific collector:
- S3CostCollector class
- collect_bucket_costs() - Per-bucket costs
- analyze_storage_classes() - Standard vs IA vs Glacier
- identify_lifecycle_opportunities() - Transition policies
- calculate_transfer_costs() - Data transfer charges
- identify_incomplete_uploads() - Cleanup opportunities

7. COST ANALYZER (src/analyzers/aws_analyzer.py)
Create comprehensive analyzer:
- AWSCostAnalyzer class
- analyze_all_services() - Aggregate all collectors
- detect_anomalies() - Unusual cost spikes (>20% change)
- calculate_waste() - Total waste across all services
- prioritize_opportunities() - Sort by savings potential
- generate_summary_report() - Executive summary

Analysis output:

```
{
 "total_monthly_cost": 120000.00,
 "total_waste": 60000.00,
 "waste_percentage": 50.0,
 "opportunities": [
  {
   "type": "spot_migration",
   "service": "EC2",
   "instance_count": 10,
   "estimated_savings": 18000.00,
   "confidence": 0.85,
   "priority": "high"
  },
  {
   "type": "idle_resource",
   "service": "RDS",
   "resource_count": 3,
   "estimated_savings": 12000.00,
   "confidence": 0.95,
```

```
      "priority": "high"
    }
  ],
  "anomalies": [
    {
      "service": "Lambda",
      "metric": "invocation_cost",
      "change_percentage": 45.0,
      "date": "2025-10-15"
    }
  ]
}
```

8. CLICKHOUSE STORAGE (src/storage/aws_metrics.py)
   Create ClickHouse integration:
   - AWSMetricsStorage class
   - store_cost_metrics() - Daily cost time-series
   - store_instance_metrics() - Per-resource metrics
   - store_optimization_opportunities() - Discovered opportunities
   - query_cost_trends() - Retrieve historical data
   - query_by_service() - Filter by service
   - query_by_region() - Filter by region

   Use existing ClickHouse tables from 0.3:
   - cost_metrics (timestamp, service, region, cost)
   - resource_metrics (timestamp, resource_id, utilization)
   - optimization_opportunities (timestamp, type, savings)

9. API ENDPOINTS (src/api/aws_costs.py)
   Create FastAPI endpoints:
   - POST /api/v1/aws/collect - Trigger collection
   - GET /api/v1/aws/costs - Retrieve cost data
     * Query params: start_date, end_date, service, region
   - GET /api/v1/aws/opportunities - Get optimization opportunities
     * Query params: min_savings, service, priority
   - GET /api/v1/aws/analysis - Get comprehensive analysis
   - POST /api/v1/aws/refresh - Force refresh from AWS

   All endpoints should:
   - Validate input with Pydantic models
   - Handle authentication/authorization

- Return standardized responses
- Log all requests
- Update Prometheus metrics

10. PROMETHEUS METRICS (update src/metrics.py)
Add AWS-specific metrics:
- aws_api_calls_total - Counter by service
- aws_api_errors_total - Counter by error type
- aws_cost_collection_duration_seconds - Histogram
- aws_total_monthly_cost_usd - Gauge by service
- aws_waste_identified_usd - Gauge by service
- aws_optimization_opportunities - Gauge by type
- aws_idle_resources_count - Gauge by service
- aws_underutilized_resources_count - Gauge by service

11. CONFIGURATION (update src/config.py)
Add AWS settings:
- AWS_ACCESS_KEY_ID - From environment
- AWS_SECRET_ACCESS_KEY - From environment
- AWS_DEFAULT_REGION - Default: us-east-1
- AWS_REGIONS - List of regions to analyze
- AWS_COST_LOOKBACK_DAYS - Default: 30
- AWS_IDLE_CPU_THRESHOLD - Default: 5%
- AWS_UNDERUTILIZED_CPU_THRESHOLD - Default: 20%
- AWS_SPOT_SAVINGS_TARGET - Default: 35%
- AWS_COLLECTION_SCHEDULE - Cron expression

12. TESTING (tests/collectors/test_aws.py)
Create comprehensive tests:
- test_aws_session_creation()
- test_cost_explorer_client()
- test_ec2_cost_collection()
- test_rds_cost_collection()
- test_lambda_cost_collection()
- test_s3_cost_collection()
- test_idle_resource_detection()
- test_spot_opportunity_identification()
- test_cost_analyzer()
- test_clickhouse_storage()
- test_api_endpoints()
- test_metrics_collection()

Use moto for AWS mocking:
- Mock Cost Explorer responses
- Mock EC2 DescribeInstances
- Mock CloudWatch GetMetricStatistics
- Mock RDS DescribeDBInstances

13. INTEGRATION TESTS (tests/integration/test_aws_integration.py)
Create E2E tests:
- test_full_collection_workflow()
- test_analysis_pipeline()
- test_storage_retrieval()
- test_api_e2e()

Requirements:
- Use real AWS sandbox account (if available)
- Or comprehensive mocking with moto
- Test error scenarios
- Test rate limiting
- Test data transformation

14. DOCUMENTATION (docs/aws-collector.md)
Create comprehensive docs:
- Setup instructions
- AWS IAM permissions required
- Configuration guide
- API reference
- Example queries
- Troubleshooting guide
- Cost Explorer API limits

TECHNICAL SPECIFICATIONS:
- Python 3.11+
- boto3 >= 1.34.0
- moto >= 4.2.0 (for testing)
- Pydantic for data validation
- asyncio for concurrent collection
- tenacity for retry logic
- Rate limiting: 400 requests/hour (Cost Explorer)
- Batch size: 100 resources per request
- Error handling: Log and continue, don't crash

- Caching: Use Redis for 1-hour cache on Cost Explorer

BEST PRACTICES:
- Use boto3 sessions, not clients directly
- Implement exponential backoff for rate limits
- Cache Cost Explorer responses (expensive API)
- Parallelize collection across regions (asyncio)
- Use pagination for large result sets
- Validate all AWS responses
- Log all API calls for debugging
- Use CloudWatch for utilization metrics (more accurate)
- Store raw data + analyzed data separately
- Update metrics after each collection

ERROR HANDLING:
- Catch boto3.exceptions.ClientError
- Handle InvalidParameterException (bad input)
- Handle ThrottlingException (rate limit)
- Handle NoCredentialsError (auth failure)
- Log errors but continue with other resources
- Retry transient errors (3 attempts)
- Skip resources that fail consistently

SECURITY:
- Never log AWS credentials
- Use IAM roles when possible (not access keys)
- Validate all input parameters
- Sanitize resource IDs before logging
- Use encrypted environment variables
- Rotate credentials regularly
- Limit API permissions to minimum required

PERFORMANCE:
- Collect costs in parallel by region (asyncio.gather)
- Use batch API calls when available
- Cache CloudWatch metric queries (expensive)
- Limit date range to avoid large responses
- Use connection pooling
- Implement request throttling

FILE STRUCTURE:

```
services/cost-agent/
├── src/
│   ├── collectors/
│   │   ├── __init__.py
│   │   └── aws/
│   │       ├── __init__.py
│   │       ├── base.py           # Base AWS collector
│   │       ├── cost_explorer.py  # Cost Explorer client
│   │       ├── ec2.py            # EC2 collector
│   │       ├── rds.py            # RDS collector
│   │       ├── lambda_costs.py   # Lambda collector
│   │       └── s3.py             # S3 collector
│   ├── analyzers/
│   │   ├── __init__.py
│   │   └── aws_analyzer.py       # Cost analyzer
│   ├── storage/
│   │   ├── __init__.py
│   │   └── aws_metrics.py        # ClickHouse storage
│   ├── api/
│   │   └── aws_costs.py          # API endpoints
│   ├── models/
│   │   └── aws_models.py         # Pydantic models
│   ├── config.py                 # Updated config
│   └── metrics.py                # Updated metrics
├── tests/
│   ├── collectors/
│   │   └── test_aws.py
│   ├── analyzers/
│   │   └── test_aws_analyzer.py
│   ├── storage/
│   │   └── test_aws_storage.py
│   ├── api/
│   │   └── test_aws_api.py
│   └── integration/
│       └── test_aws_integration.py
├── docs/
│   └── aws-collector.md
└── requirements.txt              # Updated with boto3, moto
```

VALIDATION:
After implementation:

1. Run tests: pytest tests/collectors/test_aws.py -v
2. Test API: curl -X POST http://localhost:8001/api/v1/aws/collect
3. Check metrics: curl http://localhost:8001/metrics | grep aws_
4. Verify ClickHouse: SELECT * FROM cost_metrics WHERE service = 'AmazonEC2'
5. Test analysis: curl http://localhost:8001/api/v1/aws/analysis
6. Verify Grafana: Open Cost Agent dashboard, check AWS panels

Generate complete, production-ready code with:
- All collector classes with proper error handling
- Complete API endpoints with Pydantic validation
- Comprehensive tests (unit + integration)
- Prometheus metrics integration
- ClickHouse storage layer
- Complete documentation
- AWS IAM policy examples

✅ Success Criteria

After completing this prompt, verify:

1. AWS Connection

bash

```
# Test AWS credentials
python -c "import boto3; print(boto3.client('ce', region_name='us-east-1').describe_cost_category_definition())"
# Should not raise NoCredentialsError
```

2. Cost Collection

bash

```bash
# Trigger collection
curl -X POST http://localhost:8001/api/v1/aws/collect
# Expected: {"status": "started", "job_id": "..."}

# Check status
curl http://localhost:8001/api/v1/aws/costs?start_date=2025-10-01&end_date=2025-10-31
# Expected: Cost data by service
```

3. Optimization Opportunities

```bash
# Get opportunities
curl http://localhost:8001/api/v1/aws/opportunities?min_savings=1000
# Expected: List of optimization opportunities
```

4. Metrics

```bash
# Check AWS metrics
curl http://localhost:8001/metrics | grep aws_total_monthly_cost_usd
# Expected: aws_total_monthly_cost_usd{service="AmazonEC2"} 85000.0
```

5. ClickHouse Storage

```bash
# Query stored metrics
clickhouse-client --query "SELECT service, SUM(cost) FROM cost_metrics WHERE date >= today() - 30 GROUP BY se
# Expected: Cost breakdown by service
```

6. Tests

bash

```
# Run all AWS tests
pytest tests/collectors/test_aws.py -v
# Expected: All tests pass

# Check coverage
pytest tests/collectors/test_aws.py --cov=src.collectors.aws --cov-report=term-missing
# Expected: Coverage ≥80%
```

🔗 Integration Points

The AWS Cost Collector integrates with:

1. Cost Explorer API

- Retrieves cost and usage data
- Gets forecasts and recommendations
- Accesses RI/SP utilization

2. CloudWatch API

- Collects CPU utilization metrics
- Monitors network traffic
- Tracks disk I/O

3. EC2/RDS/Lambda APIs

- Describes instances/databases/functions
- Gets resource tags
- Retrieves configuration details

4. ClickHouse (0.3)

- Stores time-series cost data
- Stores resource metrics
- Stores optimization opportunities

5. Cost Agent Metrics (0.11)

- Updates Prometheus metrics
- Tracks API calls and errors
- Monitors collection performance

6. Cost Agent Analysis (1.7)

- Feeds data to analysis engine
- Enables anomaly detection
- Supports recommendation engine

📊 Key Metrics Collected

Metric Category	Examples	Frequency
Cost Data	Total cost, cost by service, cost by region	Daily
EC2 Metrics	Instance costs, utilization, idle detection	Hourly
RDS Metrics	Database costs, connections, storage	Hourly
Lambda Metrics	Function costs, invocations, duration	Hourly
S3 Metrics	Bucket costs, storage class distribution	Daily
Opportunities	Spot eligible, rightsizing, idle resources	Daily

---

🎯 Expected Outcomes

After implementation:

- ✅ **Collect** costs from 5+ AWS services (EC2, RDS, Lambda, S3, EBS)
- ✅ **Identify** idle resources (estimated: 5-15% of resources)
- ✅ **Find** spot opportunities (estimated: 30-40% savings potential)
- ✅ **Detect** underutilized instances (estimated: 20-30% of instances)
- ✅ **Store** 30 days of historical cost data
- ✅ **Expose** AWS metrics via Prometheus
- ✅ **Provide** REST API for cost queries

📁 Files to be Created

After running this prompt, you should have:

- ☐ `src/collectors/aws/__init__.py`
- ☐ `src/collectors/aws/base.py` (~200 lines)
- ☐ `src/collectors/aws/cost_explorer.py` (~300 lines)
- ☐ `src/collectors/aws/ec2.py` (~400 lines)
- ☐ `src/collectors/aws/rds.py` (~250 lines)
- ☐ `src/collectors/aws/lambda_costs.py` (~200 lines)
- ☐ `src/collectors/aws/s3.py` (~200 lines)
- ☐ `src/analyzers/aws_analyzer.py` (~350 lines)
- ☐ `src/storage/aws_metrics.py` (~250 lines)
- ☐ `src/api/aws_costs.py` (~300 lines)
- ☐ `src/models/aws_models.py` (~150 lines)
- ☐ `tests/collectors/test_aws.py` (~500 lines)
- ☐ `tests/integration/test_aws_integration.py` (~300 lines)
- ☐ `docs/aws-collector.md` (~100 lines)

- [ ] Updated `src/config.py`
- [ ] Updated `src/metrics.py`
- [ ] Updated `requirements.txt`

Total: ~3,500 lines of new code

⏱️ Time Breakdown

Task	Estimated Time
Base collector + Cost Explorer client	30 minutes
EC2 collector	40 minutes
RDS + Lambda + S3 collectors	40 minutes
Cost analyzer	30 minutes
ClickHouse storage	20 minutes
API endpoints	25 minutes
Metrics integration	15 minutes
Unit tests	45 minutes
Integration tests	30 minutes
Documentation	20 minutes
TOTAL	**~5 hours**

Actual time with Windsurf: **25 minutes (code gen) + 20 minutes (validation) = 45 minutes**

🔄 Next Steps

After 1.2 is complete, proceed to:

NEXT: PROMPT 1.3 - GCP Cost Collector

- Similar structure to AWS collector
- Use google-cloud-billing API
- Collect GCE, Cloud SQL, Cloud Functions costs

Document Version: 1.0
Status: 🔄 Ready to Execute
Last Updated: October 21, 2025
Previous: PHASE1-1.1 (Cost Agent Skeleton)
Next: PHASE1-1.3 (GCP Cost Collector)